Staging Telephony Service Creation: A Language Approach

Fabien Latry, Julien Mercadal, and Charles Consel
INRIA / LaBRI / ENSEIRB
Department of Telecommunications
351, cours de la Libération
F-33405 Talence, France
{latry,mercadal,consel}@labri.fr

ABSTRACT

The open-endedness of telephony platforms is creating expectations among users, ranging from end-users to administrators, to create services dedicated to their activities. Not only is the population of developers heterogeneous, but the technologies underlying modern telephony range over a variety of areas such as multimedia, databases, web services, and distributed systems. This situation drastically widens the expertise required for service creation.

This paper proposes an approach to coping with the heterogeneity of both the service developers and the technologies underlying modern telephony. Our approach is based on programming languages. It consists of providing a language that is specific to each developer community with respect to its expertise (*e.g.,* programming skills) and the target application area (*e.g.,* administration). Such languages, called Domain-Specific Languages (DSLs), are organized in layers, accounting for abstraction levels.

Our layered approach to telephony service creation is illustrated by two high-level DSLs for end-user service creation, requiring no programming skills, and an expressive DSL enabling the development of expert-level telephony services. We show that layering DSLs greatly facilitates their implementation and verification of telephony-specific properties by leveraging on high-level tools.

1. INTRODUCTION

Telephony has been evolving at a frantic pace since it has converged with computer networks and multimedia. Now that telephony can interact with systems such as databases and Web services, it can offer a host of new functionalities. This situation creates a vast application area for telephony service creation, ranging from the management of telecommunications within a small business to telemarketing centers. Creating a service requires on the one hand a domain expert that has extensive telephony knowledge, like rules and practices to manage calls within an organization.

On the other hand, an implementation expert must have a thorough knowledge in a variety of areas because of the origins and combined technologies used in modern telephony. These areas include telecommunications, networking, media protocols, distributed systems, and both a telephony platform and its API. This situation creates a gap between the telephony expert, concerned with the service logic, and the implementation expert, concerned with the underlying technology components. The telephony expert seeks high-level means to express services, using modeling approaches. The implementation expert phrases solutions in terms of specific telephony platforms. This gap makes it challenging to find a correspondence between a high-level model of a telephony service and the know-how of the implementation expert, based on the existing technology building blocks.

To make the situation worse, there exist different kinds of experts on the domain and implementation sides. Indeed, telephony service creation can be done at the level of an end-user, a PABX[1] administrator, or a telephony carrier. On the implementation side, there is a number of telephony technologies that can be targeted, from the traditional PSTN[2] to the IP-based, from proprietary platforms to Java-based ones.

Nowadays, bridging the gap between domain-specific models, such as telephony models, and implementations has become a fundamental challenge that is receiving a lot of attention in the software engineering community, and in the software modeling arena, in particular.

This Paper

This paper introduces an architecture to bridge the gap between models and implementations in the domain of telephony service creation. This architecture, displayed in Figure 1, revolves around a *Domain-Specific Programming Language* (DSPL) that serves as an interface between the domain concerns and the implementation concerns. Because it is a programming language, a program written in a DSPL contains the necessary details to allow its implementation to be automatically generated in a general-purpose execution environment. Yet, it exhibits key constructs that abstract over variations of telephony platforms, facilitating its re-

[1]A Private Automatic Branch eXchange (PABX) is a telephone exchange that is owned by a private business, as opposed to one owned by a common carrier or by a telephone company.
[2]The Public Switched Telephone Network (PSTN) is the public circuit-switched telephone networks.

targeting. Because it is domain-specific, a DSPL offers high-level abstractions that are close to domain concepts. Still, a DSPL exposes abstractions that are general enough to enable various *Domain-Specific Modeling Languages* (DSMLs) to be mapped into it. DSMLs[3] provide domain experts with various degrees of abstractions to express telephony models. Because there is a variety of preferences and constraints that can be expressed by telephony experts, a variety of DSMLs can be envisioned, offering different visual or textual paradigms, and various degrees of expressivity.

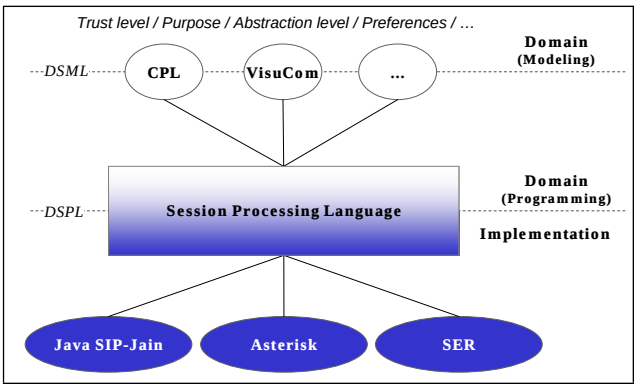


Figure 1: Bridging the gap between domain and implementation via DSPL

By separating domain and implementation concerns, our *Domain-Specific Language* (DSL) approach stages language processing, enabling specific treatments to be introduced at each layer. In a previous work, the authors presented an approach to developing compilers for DSLs, centered around the use of generative programming tools [6]. This work specifically targeted DSPLs and used aspects, annotations, and program specialization to compile the various dimensions of a DSPL. In this approach, compilation and verification of a DSL program were achieved by embedding domain-specific information into a program compiled into a general-purpose language (GPL). In doing so, they showed that the compiled program was amenable to a variety of generative programming tools. By introducing DSMLs, the abstraction level is further raised, bringing up issues and opportunities regarding the compilation of models and the verifications of domain-specific properties.

We have validated our layered DSL approach by developing a DSPL named *Session Processing Language* (SPL) [4]. This language abstracts over the details of the underlying technologies, such as telecommunications, network protocols and platform APIs. We have assessed the benefits of introducing a DSPL as a pivotal layer from both the domain and the implementation viewpoint. On the one hand, the domain-specific nature of our DSPL has greatly facilitated the implementation of two different DSMLs for telephony service creation, dedicated to non-programmers: an existing DSML named *Call Processing Language* (CPL) [23, 24] and a new one, designed by the authors, named VisuCom. This pivotal layer (DSPL) provides domain-specific programming concepts, yet hiding the underlying implementation details.

[3]In this paper, we use the term Domain-Specific Modeling Language to emphasize the modeling nature of these languages, as opposed to their programming counterpart.

We show that the nature of DSMLs makes them amenable to high-level tools that can be used to compile and verify models. On the other hand, the SPL language has been ported on two different telephony platforms, namely SIP JAIN and Asterisk – the latter port is ongoing work.

Outline

This paper is structured as follows. Section 2 analyzes the gap between programming and modeling of telephony services. Section 3 introduces our architecture based on a layered view of DSLs to bridge this gap. Section 4 and Section 5 respectively introduce a DSPL, named SPL, that abstracts over variations of telephony platforms, and two DSMLs, named CPL and VisuCom. These sections demonstrate the ability to easily introduce multiples DSMLs over the DSPL layer, and to verify domain-specific properties. Section 6 describes some related works and Section 7 concludes.

2. ANALYZING THE GAP BETWEEN TELEPHONY PROGRAMMING AND MODELING

The gap between implementations and models has long been identified and there exist approaches to tackle it. From an *implementation viewpoint*, approaches are geared toward raising the level of abstraction of the software technologies. From a *modeling viewpoint*, efforts attempt to develop mappings between abstract models into implementations. This situation applies to the telephony domain. Let us examine strategies on each side and identify their shortcomings in bridging the gap.

2.1 Programming Telephony Services

From an implementation viewpoint, bridging the gap consists of developing technology building blocks [16], such as frameworks [32], APIs and software architectures [14], that raise the level of abstraction. Experience shows that these domain-specific building blocks provide the programmer with some help. For example, a signaling protocol (*e.g.,* the Session Initiation Protocol – SIP [33]) and various media protocols (*e.g.,* the Real-Time Transport Protocol – RTP [35]) are abstracted over, using software layers and APIs. A client-server architecture, needed to cope with the distributed nature of a modern telephony platform, can be encapsulated in a framework (*e.g.,* Servlet). As well, the service logic relies on an extended library to manage a variety of call parameters (*e.g.,* caller, callee, time).

Despite all these efforts to hide the underlying intricacies, programming still incurs a tremendous overhead in a number of aspects. We illustrate these shortcomings through JAIN SIP, a standardized Java interface to SIP [10, 36], but the problem is the same with other development environments such as Microsoft Live Communication Server [26].

Large and intricate APIs.

Often, APIs are quite extensive and require a laborious and continuous learning process. Indeed, APIs are usually evolving over time, requiring steady efforts to keep existing applications in sync with the latest changes. JAIN SIP illustrates this complexity by consisting of 130 classes and more than 3000 methods, involving networking, distributed, telephony and multimedia concerns.

Lack of usability.

To facilitate re-use, methods are made generic and thus require numerous arguments that the programmer must be familiar with, as well as repetitive glue code as the prologue and epilogue of a method invocation. Furthermore, completing an elementary operation, from the domain viewpoint (*e.g.,* forwarding a call), often requires a series of tedious steps. JAIN SIP illustrates this lack of usability by including methods that may require up to 12 arguments. Furthermore, performing a domain-specific operation, like forwarding a call, may take up to 10 lines of code, containing 15 method invocations.

Lack of consistency checking.

Little, if any, checking is performed on the validity of the argument values passed to a method. No consistency checking is performed when an operation requires intermediate steps to be completed because they are glued by a general-purpose language, making it insensitive to domain constraints. For example, Java provides no support to ensure that the intermediate steps of a call redirection are successfully performed.

Platform specific.

Regardless of the abstraction level, the bottom-up approach is inherently bound to specific technology components. JAIN SIP is a telephony API that is specified as an extension of the Java platform. Similarly SIP Servlets, the SIP equivalent to HTTP Servlets, rely on Java technology. As for the Microsoft solution (Live Communication Server [26]), it offers a rich programming interface in C#.

Entangled the domain-specific logic.

Perhaps most importantly, developing a program using existing technology building blocks makes it impossible to keep the logic, expressed by the domain expert, separate from the implementation details. Instead, a program consists of an entanglement of logic and implementation elements. This situation makes it difficult for a program to evolve with the requirements of the domain expert. As can be noticed in Figure 2 (lower part), a program written in JAIN SIP entangles the service logic with various implementation details.

The above shortcomings create a gap between a domain expert and an implementation expert. Domain-specific modeling is an attempt to bridge this gap.

2.2 Modeling Telephony Services

The modeling view is best exemplified by the Domain-Specific Modeling (DSM) approach [8]. This top-down approach targets a given domain and revolves around the definition of a *Domain-Specific Modeling Language* (DSML). This language integrates the concepts and rules of the target domain, and requires the development of a code generator. Then, the developer can easily write models in domain terms, following the domain rules, and code is automatically generated [38]. An example of such a DSML is the Call Processing Language (CPL[4]) that can be used to define a range of telephony services [23, 24]. Specific restrictions guarantee domain-specific safety properties.

Although appealing by its principles, the DSM approach may have a number of shortcomings.

Too narrow.

A key idea in designing a DSML is to cover a given range of programs. This range is generally narrow enough to enable the definition of concise abstractions. Yet, if it is too narrow, it may be too limited to cover a realistic set of problems, which may ultimately compromise its usability and adoption. CPL illustrates this issue by providing a small set of operators. Doing so restricts its expressiveness. For example, CPL supports only limited call control (*e.g.*, through checks on the caller or the current time). Moreover, it offers no mechanisms for detecting and resolving conflicts among call preferences.

Too role specific.

A DSML typically targets a specific role of developer in a given domain. However, a domain often includes a variety of developer roles that may have different needs in terms of abstraction level, notations and verifications. CPL targets a unique kind of users, namely non-programmers, that develop simple services, focusing on call forwarding. Yet, other kinds of users exist in the telephony domain. For example, platform administrators require a more expressive DSML to define rich services. This situation suggests that many DSMLs are required to capture various aspects of the telephony domain.

Too high level.

Attempting to design as high-level a language as possible results in widening the gap between the domain expert and the implementation expert, adding complexity to the code generator. This situation makes the code generator hard to evolve, freezing both the language design and the target platform. CPL exemplifies this situation in that its implementation directly processes its source representation. Consequently, introducing a new parameter in the processing of a call may require changes at arbitrarily low levels of the target platform.

Too platform specific.

Overlooking the implementation aspects of a DSML may tie it to a specific platform, limiting the scope of its usage. For example, a code generator for CPL would directly produce code for a given platform. In doing so, the code generator would contain cross-cutting dependencies to the platform. As a result, porting CPL to another telephony platform would necessitate deep changes to the code generator.

This gap between a high-level language, such as CPL, and its implementation has already been identified by researchers in other domains [39, 41]. Nevertheless, there is no systematic approach to automatically translating models into an executable form.

3. A DSL-BASED APPROACH

Domain-Specific Languages (DSLs) are being studied in a lot of areas of computer science and beyond. DSLs are reported in areas such as networking [37], telecommunications [24], graphics [12], banking [11], *etc.* Generally, a DSL offers notations and abstractions that are specific to a given domain. However, a DSL may or may not have a pro-

[4]In this paper, we only consider the core of CPL as defined by the RFC 3880 [24].

gramming nature. Most DSLs developed by programming language researchers, although domain-specific, require programming skills. Examples include PADS, a language for processing data [13], and MAWL, a language for interactive Web services [20]. Hereafter, we refer to this kind of languages as Domain-Specific Programming Languages (DSPLs). The authors have mainly created DSPLs in various areas such as streaming processing [5], device drivers [25, 31], e-mail processing [7], and telephony service creation [4]. Although this work has demonstrated quantitative and qualitative benefits in terms of productivity and safety, it has mostly targeted developers having some level of programming skills, excluding non-programmer domain experts.

DSLs are created in other areas than computer science; they are generally centered around the end-user, requiring domain skills but no programming expertise. Examples include Risla, a language for describing interest rate product applications [11], and CML, a language to manage molecular information in chemistry [27]. Because of their emphasis on domain modeling, these languages should more accurately be referred to as Domain-Specific Modeling Languages (DSMLs).

As shown by the above examples, a DSL has a different meaning depending on whether it is approached from a modeling or an implementation viewpoint. A study of the literature in domain-specific modeling raises the issue of relating and distinguishing the notion of domain-specific languages from a modeling viewpoint and from a programming language viewpoint. Exploring this issue reveals the complementarity of both viewpoints; this led us to investigate this complementarity in the domain of telephony service creation.

Our Approach

We propose to introduce a layer between the domain and the implementation, representing a pivotal point between these two parts. This layer is a Domain-Specific *Programming* Language (DSPL); it permits to separate the domain concerns from the implementation concerns. In doing so, a DSML can be mapped into a *programming* yet domain-specific language, making it possible to concentrate on domain logic and abstracting over general-purpose concerns. Importantly, this approach is an enabling component to introduce a variety of DSMLs, defined with respect to a unique DSPL. Because there is a variety of preferences and constraints that can be expressed by domain experts, a variety of DSMLs can be envisioned, offering different visual or textual paradigms, and various degrees of expressivity. The development of these DSMLs is greatly simplified because they can be implemented in terms of a DSPL, making their compilation high level[5].

We have validated this approach by defining a DSPL for telephony services, named Session Processing Language (SPL) [4]. This language offers domain-specific programming constructs that permit services to be defined concisely and safely. By design, SPL guarantees critical properties that cannot be verified in general-purpose languages. On the top of SPL, various DSMLs can be defined. Specifically, we defined translations for CPL and VisuCom.

A DSPL represents an interface to the implementation concerns in that it exposes the key operations and constructs that need to be addressed when implementing it. This layering eases the re-targeting of a DSPL compiler because it is closer to a general-purpose programming and execution environment. Furthermore, this re-targeting has no impact on the DSMLs. Finally, a DSPL may be used directly by a domain programmer. Such a language allows a wide variety of applications to be developed but requires programming expertise. Alternatively, a DSPL can be hidden from the model developer and simply used as a target language to layer code generation.

We have illustrated our approach by targeting two different telephony platforms for SPL, namely SIP JAIN and Asterisk - the work on the latter platform has not been completed yet. As expected, the re-targeting has shown to have no impact on the mapping of both CPL and VisuCom into SPL.

One key benefit of our layered DSL approach is to separate concerns between the telephony domain and the implementation issues. In doing so, the semantics of DSMLs only involves domain-specific computations, making it easier to reason about models. As a result, aspects relevant to verifications can be easily extracted from models written in a DSML such as CPL or VisuCom. The nature of DSMLs makes properties to be checked more attainable by existing verification tools.

The following two sections present our approach from the implementation viewpoint and the modeling viewpoint, respectively.

4. BRIDGING THE GAP FROM AN IMPLEMENTATION VIEWPOINT VIA DSLS

The goal of a DSPL is to introduce an abstraction layer that enables the language to be implemented on a variety of telephony platforms. It should thus abstract over underlying technology building blocks such as a framework, a general-purpose programming language, libraries, *etc.* To do so, the design of a DSPL must be the result of the analysis of a variety of existing platforms. The aim of this analysis is to collect the commonalities and variations of the main existing implementations. Combined with the domain expertise, the commonalities fuel the design process of the language abstractions, while variations introduce forms of parameterization.

4.1 Implementation of a Telephony Programming Language

SPL, our DSPL for telephony service creation, have been designed to serve as an interface between telephony models and telephony platforms. Let us consider an example program written in SPL and displayed in the upper part of Figure 2. The displayed SPL fragment includes the `INVITE` method that handles incoming calls for a user named Bob. When a call occurs, it is forwarded to his current phone (Line 5, upper part). If Bob is busy, then the call is redirected to his voice mail (Line 7, upper part). Otherwise, if he cannot answer (*e.g.,* because he is absent) and the call comes from his boss, then the call is forwarded to his cell phone (Line 10, upper part).

In SPL, the response code can be seen as a simple example of an abstraction that captures the possible values returned by a request. Even though response codes are represented

[5]Our approach does not specify whether languages are compiled or interpreted. If compiled, the interpretation overhead is eliminated.

as numerical values, they are often interpreted hierarchically. To take this aspect into account, we have introduced a notation that enables such a value to be created or referred to stepwise. As an example, consider the following response code (Line 6, upper part), extracted from Figure 2: `/ERROR/CLIENT/BUSY_HERE`.

Because a DSPL is a programming language, it requires computations to be expressed explicitly, with a level of details. Yet, these computations are not expressed in an arbitrary programming language: a DSPL has specific restrictions and enrichments that permits domain-specific properties to be checked. For example, SPL has a specific Uniform Resource Identifier (URI) type that makes it possible to check the validity of an address. This strategy contrasts with languages that consider URIs as strings, preventing the destination of a call redirection from being checked.

Commonalities and variations of target platforms are not only used to design a DSPL, but they are also exploited to structure its implementation. By making explicit the commonalities of the target platforms, a DSPL eases the work of the implementer by making certain parts of a compiler invariant. For example, the hierarchical response codes of SPL are mapped into the same numerical values, regardless of the target platform. Indeed, the response codes are part of the SIP protocol specification; their values are thus guaranteed by the compliance of the target platform to the specification.

By clarifying the variations of the target platforms, a DSPL implementation distinguishes the parts that may change when targeting a new platform. Collecting these variations suggests ways of structuring an implementation to minimize the porting efforts. As an example, consider Figure 2. Its lower part displays the compiled code of the SPL program fragment, written in Java and based on a JAIN Framework. Examining the compiled code, one can observe that a number of operations are devoted to creating and testing URIs. Porting SPL requires to identify specific operations in the target implementation to manipulate URIs.

More generally, the implementation has to account for the potential target software architectures. In the case of SPL, target platforms are based on a client-server architecture. Consequently, a service, which is a single thread of computations in its SPL form, has to be split into two parts in its compiled form. Specifically, one part consists of the computations leading to sending a request (Line 5, lower part); it is implemented in Java by the `processRequest` method. In the SPL program, this part corresponds to the first line of the `INVITE` method, where a request is sent via the `forward` construct, redirecting the incoming call to Bob's current terminal (Line 5, upper part). The second part handles the response triggered by the request (Line 19, lower part); it is implemented in Java by the `processResponse` method. The request and the corresponding response are called a transaction. This mechanism takes different forms in SPL but requires a uniform compilation treatment.

4.2 Verifying Properties at the Programming Level

The telephony domain imposes stringent safety and robustness requirements. A service should not itself incur runtime errors and should respect the underlying protocol. A number of verifications can be done at the level of the DSPL layer. SPL has been designed to prevent errors that can occur when programming SIP services [4]. In doing so, verifications are factorized because they do not depend on the target platform. As a result, it becomes easier and safer to introduce a new DSML, such as CPL or VisuCom, when it is implemented on the top of a DSPL as SPL. Let us examine static verifications done by SPL.

Erroneous call processing.

A SIP service must ultimately perform some signaling actions, such that no call is lost. Furthermore, the treatment of each message must be compliant with the SIP RFC 3261 [33]. By design, SPL guarantees these two requirements. On the one hand, SPL checks that every execution path through a handler performs at least one signaling action and that these signaling actions are coherent with each other. On the other hand, SPL ensures that headers of a SIP message are manipulated in a correct way. For example, SPL prevents access to a header that is not present in the SIP message or modify a read-only header.

Erroneous state and control management.

A SIP service typically does some initial processing of a request and then forwards the request to one or more parties. Typical SIP APIs, such as JAIN SIP, separate this service logic into separate entry points for request and response processing, making it difficult to follow the service logic, and compelling the developer to manually save and restore the state that are tedious and error-prone operations. At the opposite, in SPL, request and response processing are contiguous, within a single unit, and the state is implicitly saved and restored across a `forward`. On the other hand, SPL consists of sessions that are organized into a hierarchy, with a service session at the root, the registration sessions created within the service session as its children, and dialog and subscription sessions at the leaves. The strength of SPL is to allow variables to be associated with an entire service, a registration, or a dialog, transparently managing the access to these variables across method invocations, thus ensuring that this data is manipulated in a consistent way.

Erroneous resource management.

Contrary to SIP APIs based on general-purpose languages, SPL is protected by design against infinite loops and out-of-bounds accesses to data structures or accesses to freed data. Furthermore, SPL has no mechanism for dynamically allocating data, ensuring that service execution fits within a known memory bound.

As can be noticed in Figure 2, large and intricate underlying APIs are hidden by the DSPL layer. Consequently, the upper part of the DSPL (*i.e.,* the domain) is independent of its implementation. Specifically, evolutions of an API do not impact existing models; they only affect the DSPL compiler. Moreover, the implementation experts can *componentize* the compiler to minimize the porting efforts and facilitate re-use. Finally, by bridging the gap between domain and implementation via DSPLs, programs only expose the logic without delving into the implementation details. In doing so, programming is only concerned with domain rules and domain constraints, paving the way to higher-level modeling approaches.

```
service example {
  [...]

   response incoming INVITE() {
    response r = forward 'sip:bob@phone.example.com';
    if (r == /ERROR/CLIENT/BUSY_HERE) {
      return forward 'sip:bob@voicemail.example.com';
    } else if (r == /ERROR) {
      if (FROM == 'sip:boss@example.com') {
        return forward 'tel:+19175554242';
      }
    }
    return r;
   }
 [...]
}
```

```
public class Example implements SipListener {
 [...]
 private AddressFactory factory = getAddressFactory();

 public void processRequest (RequestEvent requestEvent) {
  Request rq_request = requestEvent.getRequest();
  SipProvider rq_sipProvider = (SipProvider) requestEvent.getSource();
  String method = rq_request.getMethod();
  [...]
  if (method.equals (Request.INVITE)) {
   SipURI uri = factory.createSipURI ("bob", "phone.example.com");
   rq_request.setRequestURI (uri);
   ClientTransaction ct = rq_sipProvider.getNewClientTransaction(rq);
   ct.sendRequest (rq_request);
  }
  [...]
 }

 public void processResponse (ResponseEvent responseEvent) {
  ClientTransaction rs_ct = responseEvent.getClientTransaction();
  if (rs_ct != null) {
   Request rs_request = rs_ct.getRequest();
   Response rs_response = responseEvent.getResponse();
   SipProvider rs_sipProvider = (SipProvider) responseEvent.getSource();
   String method = rs_request.getMethod();
   rs_responseCode = rs_response.getStatusCode();
   if (method.equals (Request.INVITE)) {
    if (rs_responseCode == 486) {
     SipURI uri = factory.createSipURI ("bob", "voicemail.example.com");
     rs_request.setRequestURI (uri);
     rs_sipProvider.sendRequest (rs_request);
    } else if (rs_responseCode > 300) {
     if (rs_request.getHeader("FROM").equals("sip:boss@example.com ")) {
      TelURL tel = factory.createTelURL ("tel:+19175554242");
      rs_request.setRequestURI (tel);
      rs_sipProvider.sendRequest (rs_request);
     } else {
      rs_sipProvider.sendResponse (rs_response);
     }
    }
   }
  } else {[...]}
 }
 [...]
}
```

Figure 2: Mapping a DSPL program (SPL) into an implementation (JAIN Framework)

5. BRIDGING THE GAP FROM A MODELING VIEWPOINT VIA DSLS

By separating telephony domain and implementation concerns, the development of DSMLs is greatly simplified because they can be implemented in terms of SPL. This makes their compilation and their verification high level and amenable to existing program transformation or verification tools.

5.1 Development of Telephony Modeling Languages

A DSML enables the abstraction level to be raised further. By doing so, a DSML needs to fulfill a variety of requirements. Some telephony experts are trusted and have unlimited access to resources (*e.g.,* a telephony platform administrator, a telephony operator). Others are not trusted and can only express restricted models (*e.g.,* a secretary, an end-user). Some telephony experts may require a textual DSML to define detailed models. Others may choose a simple visual DSML to achieve fast development and evolution.

In fact, raising the level of abstraction leads to a domain-driven approach to designing a DSML, without being impeded by technology concerns. This situation results in focusing on the domain expert to provide him with a DSML that closely matches his needs. To reach this goal, one DSML does not fit all telephony experts. Therefore, we need to create variations of DSMLs that integrate parameters on the model writer such as his trust level, his purpose, his concerns, and his preferences.

If DSMLs were to be directly mapped into general-purpose execution environments, their implementation would be a major development process. Because it would be labor-intensive, few DSMLs would be developed. Furthermore, every DSML would possibly be developed independently of each other, preventing the approach from scaling up. From a software engineering viewpoint, one would like DSMLs to share layers of implementation, easing their implementation.

In our approach, this sharing is achieved by introducing the DSPL as a pivotal component between a variety of DSMLs and various implementation targets. This architecture drastically simplifies the mapping of a DSML into an implementation. We present two examples of DSMLs for telephony service creation: CPL in Figure 3 (left part) and VisuCom in Figure 4 (left part). Notice that CPL is a XML-based scripting language (left part) but script editors or graphical tools exist, making CPL accessible to non-programmers (upper part). Also, the expressivity of both languages are restricted to enable stringent safety guarantees on services. VisuCom is somewhat richer than CPL in that it makes decisions on a call based on various parameters, like a logical group that the callee belongs to[6].

The CPL service (Figure 3) corresponds to the service described previously, where a call to Bob is redirected to his voice mail, when he is busy, or to his cell phone if the call comes from his boss. The right-hand side of the figure shows the CPL service compiled in SPL. As can be noticed, the code is a straightforward translation of the CPL service.

The VisuCom service (Figure 4) is more elaborate in that, if Bob is in a meeting, a call is rejected unless it comes from the boss or a member of the Steering Committee or the Executive Committee. If so, and the call subject contains the word `important`, the call is automatically redirected to Bob's current phone and, if unsuccessful, to his cell phone. The right-hand side of the figure displays the SPL version of the VisuCom service. There as well, one can notice that the compilation is quite straightforward, even though the service is richer.

By targeting SPL, a lot of implementation intricacies are hidden (*e.g.,* SPL abstracts over the client-server model used for signaling operation), and addressed by the SPL compiler, simplifying the compilation process considerably. Indeed, translation from CPL or VisuCom to SPL only focuses on domain concerns, and so is attainable by existing tools. For example, model transformation tools such as KM3 and ATL [1, 2] have been used to rapidly develop compilation process from CPL to SPL [17, 19]. Additionally, we have realized transformations from CPL and VisuCom to SPL using the Stratego/XT program transformation environment [3, 40]. Details are not presented in this paper but excerpts of this work are available on our website [18]. By relying on a DSPL, the gap between a DSML and its implementation is reduced and therefore a DSML becomes easier to implement. As in many scripting languages, SPL could be extended with an escape to functionnalities of the low-level API. In doing so, we assure that the DSML can access arbitrary functionnalities through the underlying DSPL. Besides, a DSPL may represent a converging point for a variety of DSMLs, enabling various developer roles to be taken into account. In doing so, a developer can use a modeling paradigm specific to his role.

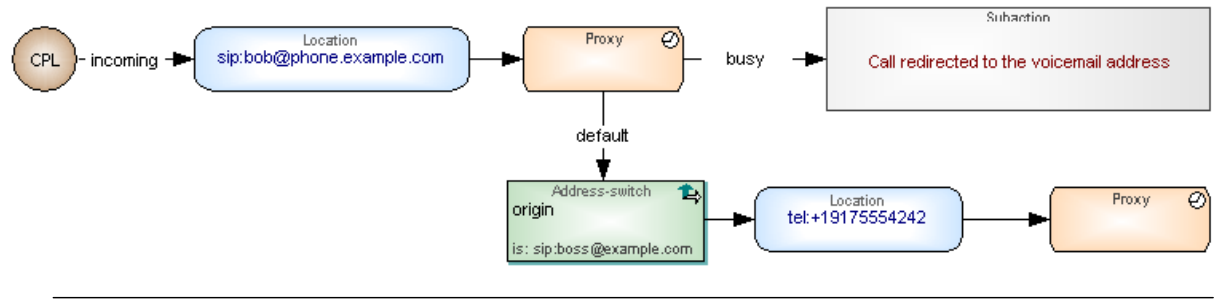
Another key benefit of the DMSL layer is to enable the verification of domain-specific properties very easily. Indeed, the semantics of DSMLs solely involves domain-specific computations which make the verification process high level and also amenable to formal verification tools. Because our DSMLs present specific restrictions, they can amount to finite-state systems. One of popular techniques for verifying finite-state systems is model-checking. In this work, we use TLA+ [22], a specification language based on temporal logic [21] and set theory, and its model checker TLC [42]. Nevertheless, other formal verification tools could be used.

Transforming CPL (or VisuCom) services into a TLA+ specification amounts to defining an abstraction that models aspects of CPL (or VisuCom) for which verification is needed. Specifically, our abstraction consists of modeling the predicates used to determine routing decisions included in a service. Again, the transformation process from a CPL (or VisuCom) service to a TLA+ specification is greatly facilitated by the nature of both CPL (or VisuCom) and TLA+. This transformation is realized with the Stratego/XT framework [18]. Let us now examine the properties to verify in telephony services.

5.2 Verifying Properties at the Modeling Level

Contrary to SPL verifications discussed in Section 4.2, properties to be verified at the level of the DMSL layer concern the language (*e.g.,* restrictions of the expressivity), but also the intention of developers. The latter is considerably simplified because a DSML, such as CPL or VisuCom, involves only domain-specific computations and therefore is easy to reason about. Domain-specific properties are expressed as TLA+ formulas and then verified by the TLC model checker [18]. For example, some features of CPL have already been identified because of their potential effects on

[6]CPL only offers the possibility to test the `sub-domain` or the `organization` of the caller.



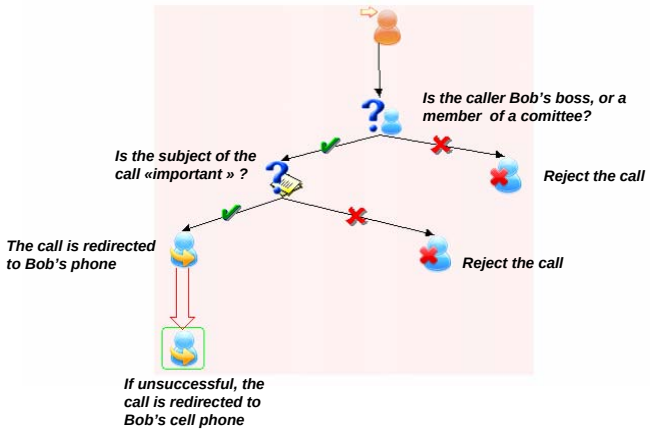

```
<?xml version="1.0" encoding="UTF-8"?>
<cpl>
 <subaction id="voicemail">
  <location url="sip:bob@voicemail.example.com">
   <redirect />
  </location>
 </subaction>
 <incoming>
  <location url="sip:bob@phone.example.com">
   <proxy>
    <busy>
      <sub ref="voicemail" />
    </busy>
    <otherwise>
      <address-switch field="origin">
        <address is="sip:boss@example.com">
         <location url="tel:+19175554242">
          <proxy />
         </location>
        </address>
      </address-switch>
    </otherwise>
   </proxy>
  </location>
 </incoming>
</cpl>
```

```
service example {
 processing {

  dialog {

   response incoming INVITE() {
    response r = forward 'sip:bob@phone.example.com';
    if (r == /ERROR/CLIENT/BUSY_HERE) {
      return forward 'sip:bob@voicemail.example.com';
    } else if (r == /ERROR) {
      if (FROM == 'sip:boss@example.com') {
        return forward 'tel:+19175554242';
      }
    }
    return r;
   }
  }
 }
}
```

Figure 3: Mapping a DSML (CPL) into a DSPL (SPL)




```
service complexe {

  local uri<> getAddresses (String group);

  processing {
   uri<> add1 = getAddresses("Steering Committee");
   uri<> add2 = getAddresses("Executive Committee");
   uri<> addresses = <'sip:boss@example.com'> # add1 # add2;

    dialog {

     response incoming INVITE() {
      foreach (address in addresses) {
       if ((FROM == address) && (SUBJECT == "Important")) {
        response r = forward 'sip:bob@phone.example.com';
        if (r == /ERROR) {
          return forward 'tel:+19175554242';
        }
       }
      }
      return /ERROR with { reason = "In Meeting" }
     }
    }
  }
}
```

Figure 4: Mapping a DSML (VisuCom) into a DSPL (SPL)

the consistency and reliability of services written by non-expert developers [28]. Consequently, it appears very important to verify some domain-specific properties related to the service context (*e.g.*, service interaction, caller). We now detail examples of such properties.

No duplicate redirect.

This property ensures that an execution path does not contain more than one redirection to the same address. In doing so, not only are unnecessary operations detected, but the service execution time is also kept to a minimum.

No redirect to the caller.

This property ensures that an execution path does not redirect the call to the caller. For example, a telephony service should not redirect the call to Bob if the caller is Bob. Generally, this situation is not intended by the service developer because it implicitly prevents Bob from talking to the service client.

No infeasible path.

This property ensures that a cascade of tests is correct in that it is feasible. Two kinds of tests are considered: (1) addresses, that is, caller identifiers and (2) dates. Consider an example of the former kind, displayed in Figure 5. This service is erroneous in that, when a call occurs, if the sender's address comes from the sub-domain `example.com` (node 2), then the call is forwarded to Bob's cell phone (node 3). Otherwise, if the sender's address comes from the sub-domain `work.example.com` (node 5), then the call is redirected to his voice mail (node 6). Nevertheless, this case can never happen because it is subsumed by the first test (node 2). Such a cascade of address tests is common in realistic services, creating a need to verify its feasibility.

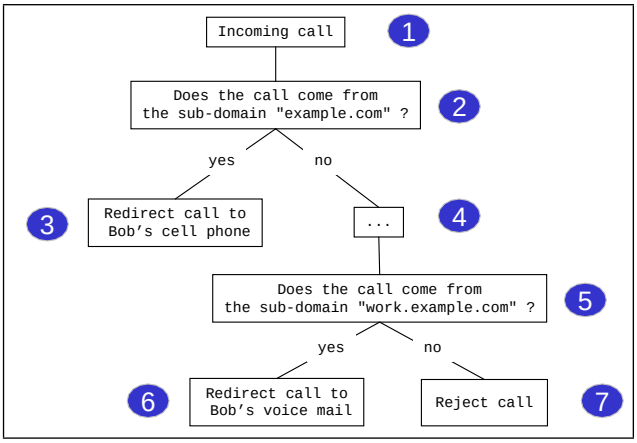


Figure 5: Erroneous CPL service (sender's address)

The second part of this property guarantees that a cascade of tests on calendar events are feasible. An example of an infeasible execution path is displayed in Figure 6. This service treats calls differently depending on the day of the week, that is, whether a call occurs on Tuesday. If so, it tests whether the call occurs between 07/12 and 07/17. However, there is no Tuesday within that period of time, therefore this last execution path is not feasible.

Because of their decision-tree nature, telephony services include cascades of tests that involve predicates on a variety of aspects. The above examples demonstrate the interest of verifying that a service does not include infeasible paths.

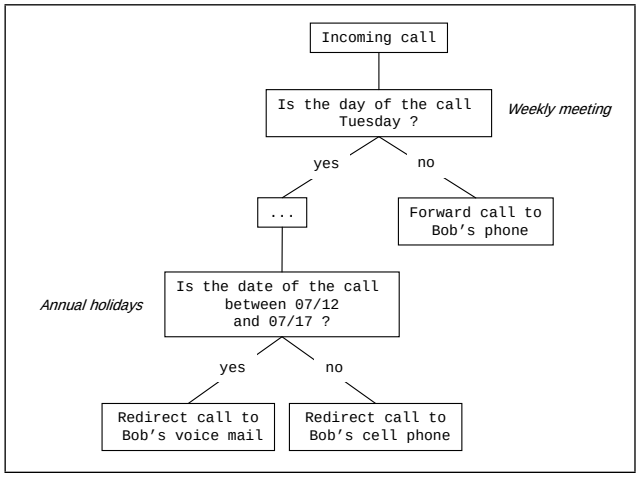


Figure 6: Erroneous CPL service (date of the call)

Service interaction.

This property ensures that a call will not be infinitely redirected from one CPL (or VisuCom) service to an other. Let us consider three telephony services for *Alice*, *Bob* and *Charlie* respectively. *Alice*'s service redirects all the calls to *Bob*. *Bob*'s service redirects all the calls to *Charlie*. *Charlie*'s service redirects all the calls to *Alice*. As a result, if someone wants to contact *Alice*, *Bob* or *Charlie*, the call will loop without never being taken.

If one of the above properties is not verified by a CPL (or VisuCom) service then the TLC model checker generates a counterexample.

The above verifications could have be realized at the level of SPL. However, CPL exposes this information at a level that makes its translation to a TLA+ specification straightforward. SPL is low level, and abstracts certain implementation details (*e.g.,* the SIP protocol) that therefore must be modeled too. As a result, the step of transformation between SPL and TLA+ would have been more important.

Finally, the domain-specific and high-level nature of DSMLs enables such domain-specific properties to be easily formulated and implemented into existing verification tools such as TLA+.

6. RELATED WORK

The DSL approach is being actively studied because of its potential to greatly improve software development. As a result, a lot of approaches and tools have been proposed to develop DSLs, addressing both their design and implementation.

Concerning the design, a number of frameworks and development environments are available such as MetaCase [38], AMMA (Atlas Model Management Architecture) [1], and Microsoft's initiative on Software Factories [15]. These approaches consist of a rich toolkit for creating DSLs. It offers graphical environments, to design visual languages, powerful mechanisms, to constrain the composition of language constructs, and a high level of automation, to generate tools

required by a new DSL.

Nevertheless, these tools offer limited support for code generation. More specifically, DSLs often introduce a large gap between models and implementations, requiring the intervention of implementation experts to cover all aspects of code generation. The layered solution, presented in this paper, is complementary to DSL development environments: it provides a staged approach that makes the code generation process amenable to program generation.

Concerning the implementation of DSLs, numerous approaches have emerged. Following the classification of model transformation approaches introduced by Czarnecki [9], we identify four main strategies to implement DSLs: code generation, semantic-based compiler generation, model-based transformation, and XML-based transformation.

Code generation.

Developing a DSL-specific code generator is the main alternative to our approach. It consists of a direct compilation from a DSL to some code. This solution, promoted by the model-driven development approach, presents a number of disadvantages. Firstly, the DSL compiler is often developed as a monolithic program, interweaving implementation and domain concerns. This results in a costly and complicated software development that exposes few opportunities for reuse. Because of the entanglement of domain and implementation aspects, domain experts cannot introduce new functionalities without an extended knowledge of the code generator. This situation makes it difficult for a DSL to evolve with the requirements of the domain expert.

In our approach, the code generation process is split into two phases: compiling a DSML and compiling a DSPL. The DSML compiler solely concentrates on domain-specific computations. New functionalities can be introduced at this stage without being concerned with implementation issues. The DSPL compiler focuses on implementation concerns, handling low-level details (*e.g.,* platforms, frameworks and protocols).

Semantic-based compiler generation.

The goal of this approach is to generate a compiler from a formal specification of a language. Denotational semantics is a prime example of this approach; it has been used in a number of compiler generators [34]. Semantic-based compiler generation has mainly been used to generate compilers for general-purpose languages. In this context, this approach has had to compete with hand-crafted compilers, without much success. As a result, little research is still done on this topic nowadays.

Model-based transformation.

This approach assumes that a DSL is a high-level language and thus is well-suited for model transformation tools like KM3 and ATL [1]. The idea is to define a metamodel for such a DSL and to apply transformations to translate it into another form. In this regard, model-based transformation is complementary to our approach in that it offers additional tools to manipulate models written in DSMLs.

XML-based transformation.

The goal of this approach is to use XML transformers to compile a DSML model into a DSPL program. To do so, the source program is assumed to be represented in XML. Tools like XPath [29] and XLST [30] can be used for this purpose. Nevertheless, some languages may not fit well with XML and lead to the development of cumbersome transformation processes. This is due to the fact that XML tools are ultimately dedicated to data conversion, not program transformation.

As can be noticed, our DSL layered approach is complementary to existing approaches. By introducing both DSMLs and DSPLs, it allows language processing to be staged. It is also complementary to rich graphical environment that deals with the design of visual languages. Moreover, by separating domain and implementation, our DSL approach enables specific treatments to be introduced at each layer. Thus, the DSPL layer is devoted to deal with the compilation of implementation details, whereas the DSML layer is dedicated to the logic translation. Finally, by introducing DSMLs, the abstraction level is further raised, bringing up issues and opportunities regarding the compilation of models and the verifications of domain-specific properties, for example.

7. CONCLUSIONS

This paper presents an approach to bridging the gap between the telephony domain and the telephony platforms, making service creation accessible to domain experts, without requiring implementation expertise. Our approach is supported by two layers of DSLs where DSMLs are dedicated to the creation of a class of services and a DSPL is an interface between telephony experts and implementation experts. This layering enables DSML compilation to focus on aspects related to the target class of services, deferring the intricacies of target platforms to the DSPL compiler. In doing so, the development of DSMLs is facilitated by making their compilation amenable to program transformation tools. Furthermore, DSML compilers become independent of a given platform.

Because the semantics of a DSML only involves telephony-specific computations, properties can be more easily verified in models. Domain-specific models can be translated into specifications to be processed by a verification tool, guaranteeing domain-specific properties.

We have validated our approach with two DSMLs, namely CPL and VisuCom; they provide domain-specific notations and abstractions to create telephony services. These DSMLs allow domain experts to directly express solutions in domain terms, without necessarily requiring programming skills. We used a DSPL named SPL as an interface between the domain and the implementation layer. SPL is a DSL that abstracts over underlying telecommunication technologies but requires programming expertise. These languages illustrate the high-level nature of the transformation process and demonstrate how much DSMLs can leverage existing tools for compilation and domain-specific properties verification.

8. REFERENCES

[1] J. Bézivin, G. Hillairet, F. Jouault, I. Kurtev, and W. Piers. Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In *Proceedings of the International Workshop on Software Factories at OOPSLA'05*, San Diego, California, USA, 2005.

[2] J. Bézivin, F. Jouault, and D. Touzet. An Introduction to the ATLAS Model Management Architecture. Technical Report 05-01, LINA, 2005.

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: Components for Transformation Systems. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation (PEPM'06)*, pages 95–99, New York, NY, USA, 2006. ACM Press.

[4] L. Burgy, C. Consel, F. Latry, J. Lawall, L. Réveillère, and N. Palix. Language Technology for Internet-Telephony Service Creation. In *Proceedings of the IEEE International Conference on Communications (ICC'06)*, Istanbul, Turkey, 2006.

[5] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. Spidle: A DSL Approach to Specifying Streaming Application. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03)*, Erfurt, Germany, September 2003.

[6] C. Consel, F. Latry, L. Réveillère, and P. Cointe. A Generative Programming Approach to Developing DSL Compilers. In R. Gluck and M. Lowry, editors, *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *Lecture Notes in Computer Science*, pages 29–46, Tallinn, Estonia, September 2005. Springer-Verlag.

[7] C. Consel and L. Réveillère. A Programmable Client-Server Model: Robust Extensibility via DSLs. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 70–79, Montréal, Canada, November 2003. IEEE Computer Society Press.

[8] S. Cook. Domain-Specific Modeling and Model Driven Architecture. In *The MDA Journal: Model Driven Architecture Straight from the Masters*, chapter 3. D. Frankel and J. Parodi edition, December 2004.

[9] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, USA, 2003.

[10] J. Deruelle, M. Ranganathan, and D. Montgomery. Programmable Active Services for JAIN SIP. Technical report, National Institute of Standards and Technology (NIST), June 2004.

[11] A. van Deursen. Domain-Specific Languages versus Object-Oriented Frameworks: A Financial Engineering Case Study. In *Proceedings of Smalltalk and Java in Industry and Academia (STJA'97)*, pages 35–39. Ilmenau Technical University, 1997.

[12] Conal E. An Embedded Modeling Language Approach to Interactive 3D and Multimedia Animation. *Software Engineering*, 25(3):291–308, 1999.

[13] K. Fisher and R. Gruber. PADS: A Domain-Specific Language for Processing Ad-Hoc Data. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pages 295–304, Chicago, IL, USA, 2005. ACM.

[14] D. Garlan and M. Shaw. *An Introduction to Software Architecture*, volume I. World Scientific Publishing Company, Singapore, 1993.

[15] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons, 2004.

[16] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Addison-Wesley Professional Computing Series. Addison-Wesley, September 2004.

[17] ATLAS group and Phoenix group. ATL Use Case - DSLs Coordination for Telephony. http://www.eclipse.org/m2m/atl/usecases/DSLsTelephony/, 2006.

[18] Phoenix group. Processing Domain-Specific Modeling Languages - Compilation (Stratego/XT) and Verifications (TLA+). http://phoenix.labri.fr/processings, 2006.

[19] F. Jouault, J. Bézivin, C. Consel, I. Kurtev, and F. Latry. Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. In *Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD'06)*, Nantes, France, July 2006.

[20] D.A. Ladd and J.C. Ramming. Programming the Web: An application-oriented language for hypermedia service programming. In *Proceedings of the 4th International World Wide Web Conference*, Boston, Massachusetts, December 1995.

[21] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[22] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[23] J. Lennox. *Services for Internet Telephony*. PhD thesis, Columbia University, January 2004.

[24] J. Lennox and H. Schulzrinne. Call Processing Language (CPL): A Language for User Control of Internet Telephony Services. Request For Comments (RFC) 3880, The Internet Engineering Task Force (IETF), October 2004.

[25] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI'00)*, pages 17–30, San Diego, California, October 2000.

[26] Microsoft. *Live Communications Server 2005 Enterprise Edition Deployment Guide*, November 2004.

[27] P. Murray-Rust. Chemical Markup Language (CML). *World Wide Web Journal*, 2(4):135–147, 1997.

[28] M. Nakamura, P. Leelaprute, K. Matsumoto, and T. Kikuno. Semantic Warnings and Feature Interaction in Call Processing Language on Internet Telephony. In *Proceedings of the 2003 Symposium on Applications and the Internet (SAINT'03)*, pages 283–291, Washington, DC, USA, 2003. IEEE Computer Society.

[29] W3C Recommendation. XML Path Language Version 1.0. http://www.w3.org/TR/xpath, November 1999.

[30] W3C Recommendation. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, November 1999.

[31] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. A DSL Approach to Improve Productivity and Safety in Device Drivers Development. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 101–109, Grenoble, France, September 2000. IEEE Computer Society Press.

[32] D. Roberts and R. Johnson. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In *Proceedings of Pattern Languages of Programs (PLoP'96)*, Allerton Park, Illinois, September 1996.

[33] Rosenberg, J. et al. SIP : Session Initiation Protocol. Request For Comments (RFC) 3261, The Internet Engineering Task Force (IETF), June 2002.

[34] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[35] Schulzrinne, H. et al. RTP: A Transport Protocol for Real-Time Applications. Request For Comments (RFC) 1889, The Internet Engineering Task Force (IETF), January 1996.

[36] Sun Microsystems. The JAIN SIP API Specification v1.1. Technical report, Sun Microsystems, June 2003.

[37] S. Thibault, C. Consel, and G. Muller. Safe and Efficient Active Network Programming. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, 1998.

[38] J. P. Tolvanen. MetaEdit+: Domain-Specific Modeling for Full Code Generation Demonstrated [GPCE]. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 39–40, New York, NY, USA, 2004. ACM Press.

[39] J.P. Tolvanen and S. Kelly. Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. In *Proceedings of the Software Product Line Conference (SPLC'05)*, volume 3714 of *Lecture Notes in Computer Science*, pages 198–209. Springer, 2005.

[40] E. Visser. Meta-Programming with Concrete Object Syntax. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[41] H. Wada and J. Suzuki. Modeling Turnpike Frontend System: A Model-Driven Development Framework Leveraging UML Metamodeling and Attribute-Oriented Programming. In *Proceedings of the 8th Model Driven Engineering Languages and Systems (MoDELS'05)*, volume 3713 of *Lecture Notes in Computer Science*, pages 584–600. Springer, 2005.

[42] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, pages 54–66, London, UK, 1999. Springer-Verlag.